A TIME SHARING OPERATING SYSTEM FOR TDC - 316

by

PARITOSH K. PANDYA

COMPUTER SCIENCE PROGRAMME

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JULY, 1982

A TIME SHARING OPERATING SYSTEM FOR TDC-316

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by

PARITOSH K. PANDYA

to the
COMPUTER SCIENCE PROGRAMME
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
JULY, 1982

CERTIFICATE

This is to certify that the thesis entitled "A TIME SHARING OPERATING SYSTEM FOR TDC-316" has been carried out by Sri Paritosh K. Pandya under my supervision and has not been submitted elsewhere for the award of a degree.

A.S. Sethi
Assistant Professor
Computer Science Programme
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

Kanpur
July 1982

ACKNOWLEDGEMENT

I wish to express my gratitude to Dr. A.S. Sethi for his inspiring guidance throughout the course of this thesis. He has been very kind in sparing his valuable time for regular fruitful discussions.

I am thankful to Aarti Kumar for her excellent team-spirit.

I am very greatly indebted to George Paul for his ready help at all times.

I am also very thankful to Kireeti Kompella and K.S. Satish for correcting the manuscripts. My special thanks to Harish Karnick for his suggestions about the thesis.

Finally, I must thank Sri Mohammed Anwar for elegant typing.

Kanpur
July 1982

- Paritosh K. Pandya

ABSTRACT

This thesis describes the design and implementation of a Time Sharing Operating System for the TDC-316 Computer.

The operating system is designed as a hierarchy of layers. Kernel, the innermost layer, provides the processor allocation and memory management for a fixed number of processes. The next layer contains the device subsystems, which provide logical services from the devices. This layer also contains the Error-trap handlers and various system processes. The top layer consists of the Monitor Call Service Routine. This routine handles the users' requests for system services. The filing system runs in a simulated supervisor mode.

CONTENTS

CHAPTER 1

INTRODUCTION

"Program development and execution are two radically different functions. Much can be obtained by assigning each function to a computer best suited for it." [DOLO 78]

Thus we can have a large central computer connected in a network to many small dedicated computers, providing "Development facilities" to the users. Most of the large processing occurs on the central computer and most of the large software also recides and executes on the central computer.

The development systems locally provide users with following utilities:

(1) A powerful filing system to maintain user files.

(2) Good tools for manipulation of text, documents and programs; e.g.

(a) Editors

(b) Text formatters

(c) Documentation preparation systems

(3) Terminal oriented system, which allows interactive use of the development facilities.

(4) Facilites to maintain and transfer files on many different media like removable disks, magnetic tapes, cards, paper tapes etc., and facility to print text.

(5) Facility to carry out large processing over the central computer.

Most simple systems allow this processing to be carried out only in a batch mode by remote job entry. However modern network technology also allows an interactive use of the central computer.

Such a system will have many advantages.

(1) The development system can be implemented on a computer of moderate size. We can have many such development systems connected to the central computer, increasing the availability of computing facility to a much larger number of users. Such facility is best suited for computer centre environments, where computer availability is difficult.

(2) Separating program development and processing functions leads to a much more homogeneous workload to both the central computer and the development system.

The central computer now mainly carries out large compute-bound processing. It can be tuned to give larger throughput for such tasks.

Most of the processing carried out by the development system is a form of text processing and text manipulation functions. The development system can be designed for

these workload characteristics.

(3) In university environment, the development system can be tuned to the necessities of novice programmers; e.g. it may implement a convenient single language environment for a basic programming course.

A development system presents a very visible environment to the users (unlike the front ends), and thus has very stringent requirements on

(1) User response time.

(2) System reliability

(3) Security and protection between the users

(4) System availability

1.1 THE PWS-316 SYSTEM

The PWS-316 (Programmer's Work Station*) project is an implementation of a development system on TDC-316 computer. The TDC-316 computer at I.I.T. Kanpur is connected to a central DEC-10 system via an asynchronous line. A high bandwidth synchronous line is also being installed for fast transfer of data between the two machines.

The PWS-316 is intended to support many simultaneous users in time-sharing mode. The project is divided into three phases.

---

* The name is adapted from the Programmer's Work Bench implemented at Bell Labs. [DOLO 78].

(1) The design and implementation of a time-sharing operating system to support many online users.

(2) The design and implementation of network software for setting up communication between TDC-316 and DEC-10 systems.

(3) The development of utility programs for supporting program development.

This thesis concerns itself with the design and implementation of the basic time-sharing operating system required for running the PWS-316 system.

An associated project implements the filing system supported by this operating system. [KUMA80]. These two projects complete the phase 1 of the PWS-316 project.

Chapter 2 of this thesis gives the hardware description of the TDC-316 machine.

Chapter 3 gives the user's view of the PWS-316 system and defines the virtual machine implemented by the operating system. It also gives an overview of the structure and working of the operating system.

Chapter 4 gives the details of the operating system design.

Chapter 5 describes the tools used and the methodology followed in the project implementation. It also describes the testing.

Chapter 6 gives the designer's comments about the designed operating system and proposes some extensions to the design.

CHAPTER 2

HARDWARE DESCRIPTION

The PWS-316 project is implemented on a 16 bit mini-computer, TDC-316. The TDC-316 system comprises of a CPU, main memory, relocation hardware, a system clock and various peripheral equipment. Fig. 2.1 gives a block diagram of the TDC-316 system. This chapter briefly describes the relevent features of the available hardware.

The CPU is a 16 bit bipolar machine. It contains a Program Counter (PC), a Processor Status Word (PSW) and 14 General Purpose Registers (GPRs). The processor supports two modes, Privileged and Non-privileged. Some instructions are executable only in privileged mode. Each mode has its own Stack Pointer (SP). The remaining 12 GPR's are divided into two sets of 6 registers each. At a time only one set can be accessed, depending upon a bit-setting in the PSW. In PWS-316 system, the operating system runs in privileged mode using one set of GPR's (called Kernel Register Set). The user programs execute in non-privileged mode and use the other register set (called User Register Set).

The CPU has a priority interrupt mechanism with 8 priority levels. Each device is connected to the system at

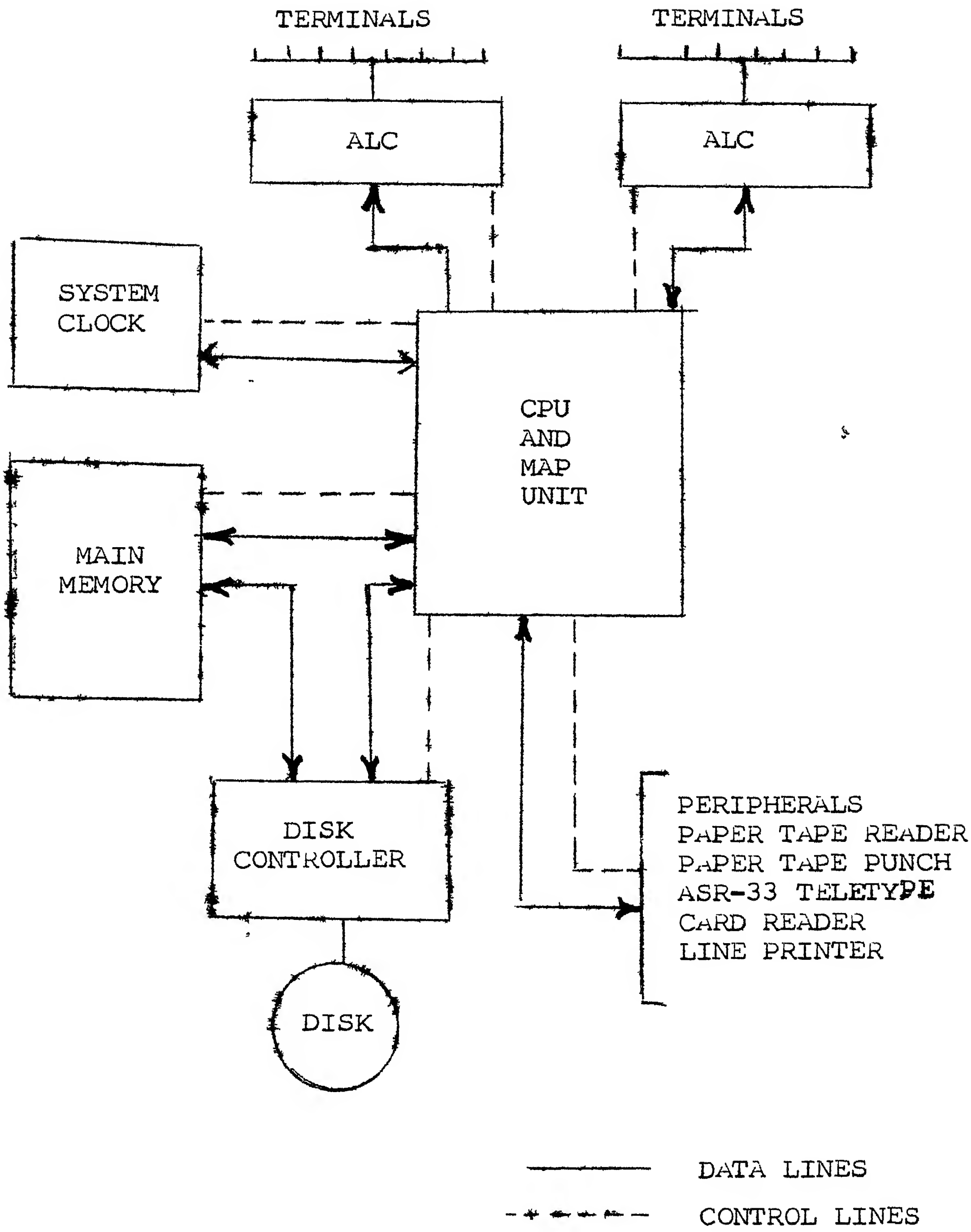


FIGURE 2.1 HARDWARE CONFIGURATION

one of these priority levels and has its own interrupt vector. The CPU can be assigned any priority by setting bits in PSW under program control. A device interrupt is enabled for processor interruption only if the device has a higher priority than the current priority of the CPU. All the control and data registers for the devices are addressable as memory locations [TDC 1].

The directly addressable memory on TDC-316 system is 64K Bytes. This can be accessed either as a byte, or as a word of two bytes. The system provides a Memory Allocation and Protection (MAP) unit which allows extending the physical memory space to 128K words. The physical memory is divided into 32 Word blocks.

It is possible using the MAP unit, to keep more than one program in the main memory simultaneously. Each program has a virtual address space of 8 Instruction segments and 8 Data segments. Each segment can have a maximum length of 4K Words (in steps of 32 Words).

The hardware contains one MAP Register for each Virtual segment. By loading these registers with appropriate values, it is possible to relocate each segment at any block boundary. It is also possible to specify the access protection for each segment as no access, read only access or read write access. Any illegal memory access results in a Map-abort trap.

The hardware contains two sets of Map registers. One set, called Supervisor Map Registers is used to relocate the

the memory references in privileged mode. This is used to run the operating system. The other set, called User Map Registers is used to relocate the memory references in non-privileged mode and is used to run user programs. Thus MAP unit allows facility to isolate the users from each other and from the operating system. It also allows the system to allocate the memory in discontinuous parts, and to implement swapping using the memory faults TDC 2 . The present configuration of TDC-316 contains 32K Words of core memory.

The system has a clock unit, which consists of an ELAPSE TIMER (ET) and a TIME-OF-DAY counter. The Elapse Timer can be programmed to interrupt the CPU at regular intervals of 1 micro-second to 2 ** 27 milli-seconds. The TIME-OF-DAY counter, incremented every milli-second, is used to maintain a realtime clock [TDC 3].

The system has a number of peripheral devices connected to it.

The DISK CONTROLLER allows upto four DISK DRIVES to be connected to the system. The controller accepts commands to transfer a sector of data between the main memory and the disk. The transfer is done in Direct Memory Access (DMA) mode and its completion is indicated by an interrupt. The present configuration contains a single Disk Drive having 7.25 Mbytes storage capacity [TDC 3].

All the Video Terminals are interfaced to the system through Asynchronous Line Controllers (ALCs). Each ALC supports eight terminals (or lines). Using these controllers, characters can be transmitted and received from all the terminals in parallel, in full duplex mode. The character to be transmitted is put into the transmit buffer, along with the desired line number. The completion of transmission is indicated by receipt of an ACK character from that line. The ALC interrupts the CPU after either receiving eight characters or within 3.3 milli-sec's of receiving one character. The present configuration has two sets of Asynchronous Line Controllers.

Besides these the system has all the standard peripherals listed below.

(1) A line printer with printing speed of 300 lines per minute.

(2) A Card Reader reading 300 cards per minute.

(3) A console teletype.

(4) High speed paper tape punch.

(5) High speed paper tape reader.

The details of these devices can be found in [TDC 3].

CHAPTER 3

3.1 USER'S VIEW OF THE PWS-316 SYSTEM

PWS-316 Development System provides users with some basic facilities for composing, editing and maintaining their programs and text. It acts as a Programmer's Work Station.

Each user interacts with the system through a video terminal. The user obtains the use of various utilities by a set of commands. A command processor program accepts these commands and starts execution of the appropriate utility program.

The user is provided with a powerful filing facility. The filing system* has a hierarchical structure, allowing any level of Subfile Directories (SFDs). Thus the file directories form a tree structure. The user can specify a file in the system by its path from the root of the directory tree. Each user has a Current Directory Pointer (CDP), which he can set to point to any directory file in the tree (subject to protection checks). The use can also specify

---

* The filing system was implemented by Aarti Kumar at I.I.T. Kanpur [KUMA 82].

a file as a path starting from the CDP. Further, the filing system also provides a comprehensive protection scheme based on Owner, Project partner, Others scheme. [RITC 74]

The user is provided with commands to list directories, to type files, to copy files and to carry out other such functions.

A line-editor is also available to the user for editing files and for creating new files.

User can give his text files for printing on a line-printer. A line-printer spooler prints these files in sequence.

The system also provides commands for transferring files between the TDC-316 and the DEC-10 systems, and for submitting large processing tasks to the DEC-10 system in Batch mode.

Besides these, the user can carry out various system functions like logging into the system, logging out and getting system information.

The system recognises a privileged user by his Project, Programmer number. The privileged user is allowed various system management functions. These include creating new users, deleting users, changing accounting information and introducing new utilities in the system. This user is also allowed access to all the files in the system.

3.2 THE VIRTUAL MACHINE

PWS-316 supports multiple online users in time-sharing mode. These users share the system resources like filing space, memory, CPU, devices etc.

To manage these resources efficiently and to isolate the users from each other, a time sharing operating system is implemented.

This operating system presents a virtual machine to the utility Programs run by each user. The virtual machine presents a logical view of the various resources and devices to the user program which is

(1) Simple to use.

(2) Hides all the resource sharing aspects from the user. The user is not allowed any direct control over the devices and resources.

(3) Protects users from each other.

This section describes the various aspects of the virtual machine defined by the PWS-316 operating system.

3.2.1 USER VIRTUAL SPACE

The PWS-316 memory system supports segmentation with static linking [BENS 69].

A segment is a logical collection of information (e.g. a Library of Subroutines). It is the basic entity in terms of which this operating system manages memory.

The user's virtual space consists of an I-space and a D-space. All the instruction fetches go to the I-space, while all the data fetches go to the D-space. I-space contains only pure code.

Both the I-space and the D-space are divided in 8 segments each, called code-segments and data-segments respectively. The code-segments are pure, and can be shared by all programs. The data-segments can be read only or read-write. The read-only data-segments contain constant data initialised at the time of loading.

The above scheme has a number of advantages.

(1) I-spaces of different programs can share code-segments containing commonly used routines.

(2) Different users executing the same program can share the same copy of the program's code. Similarly the read-only data-segments can also be shared by users.

(3) Segments can be "posted" from one User's virtual space to another user's virtual space for transferring a large amount of data, efficiently.

(4) Segments form a natural unit in terms of which memory allocation and swapping may be done. Even implementation of overlay segments becomes quite straight forward.

(5) If swapping is implemented, the pure code-segments do not have to be swapped out.

(6) The above scheme lends itself to a direct implementation on the TDC-316 MAP unit, which provides relocation and protection hardware supporting I-space segments and 8 D-space segments.

In PWS-316 operating system, a program can use data-segments 0 to 5 for its own data. The segment 0 is called the Communication-Segment, and is shared by the user, the operating system and the supervisor for passing large amount of data (e.g. a file path specification). This segment is also used for passing arguments from one program to another, in program-chaining.

The data-segment 6 is called the Stack-Segment and is used to maintain the user stack. When a new program starts execution, the system resets the stack pointer to the top of the stack-segment.

The data-segment 7 contains various device registers and is not accessible to the user program.

In PWS-316 system, most of the utilities are written as sharable programs and a single copy of their code is kept in the main memory. This is shared by all users. The read-only data-segments are also shared. Figure 3.1 shows the schematic diagram of a user's virtual space.

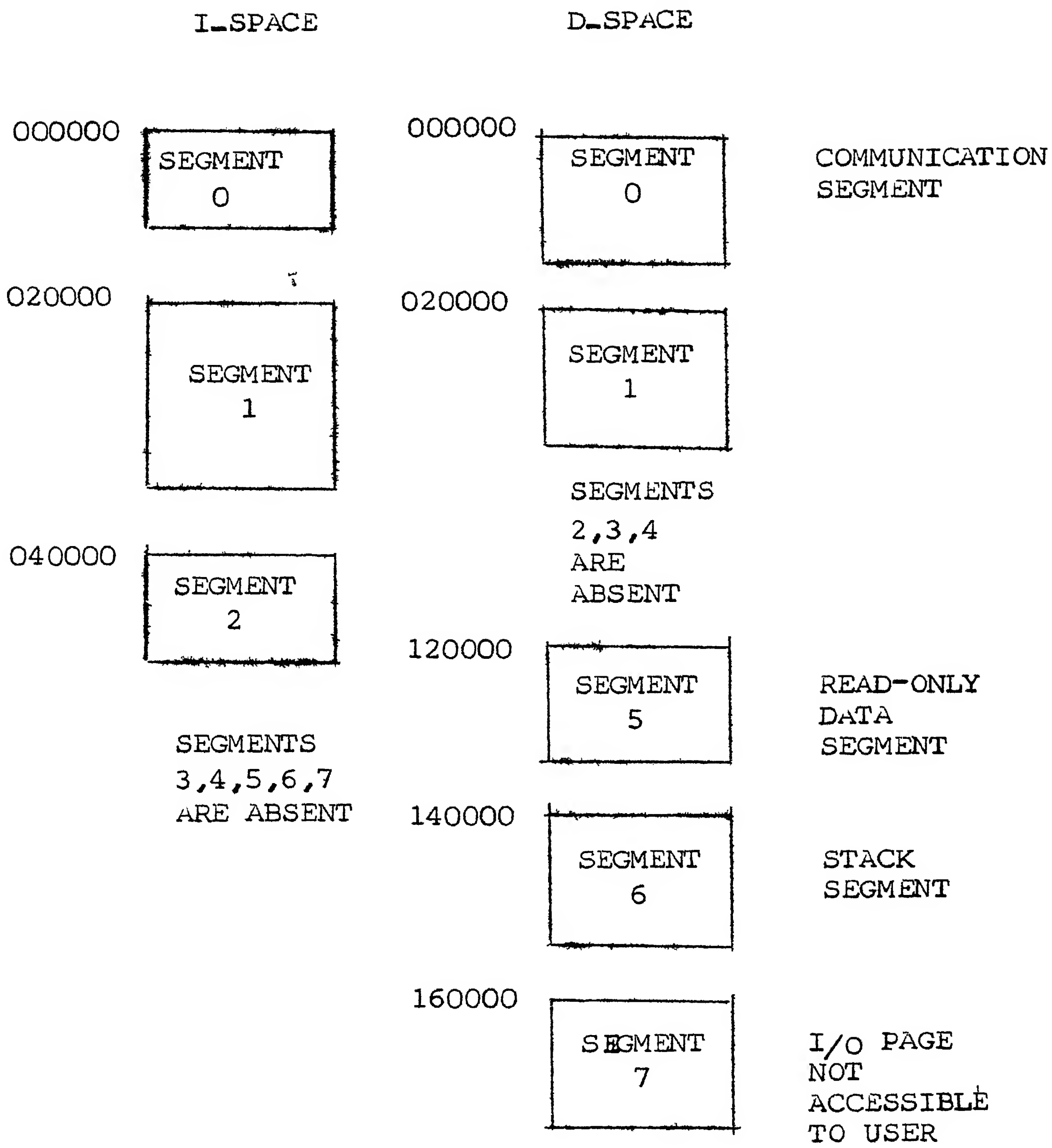


FIGURE 3.1 A TYPICAL USER VIRTUAL SPACE

The user is provided with a linking facility to divide his programs into logical segments (Chapter 5).

3.2.2 USER INSTRUCTION SET AND MONITOR CALLS

User programs are allowed the use of all non-privileged instructions in the instruction set of TDC-316 machine [TDC 1]. User programs run in non-privileged mode, using the register set 1.

User programs obtain the system services by making Monitor Calls to the operating system.

A monitor call is made by putting the call number in the user register R7 and by executing a TRAP instruction. The arguments to the monitor call are passed through the user registers. Long arguments are passed by putting them in the communication-segment, and by passing a pointer to them in a user register. Results of the monitor call are also returned in the user registers and through communication-segment.

The user program is expected to save the values of the registers used in a monitor call. These values can then be restored after the monitor call. System provides a macro-library for performing these functions (Appendix A). These macros perform the required register saving, monitor call linkage and restoring of the registers.

3.2.3 PROGRAM CHAINING

The virtual machine provides facility for CHAINING programs. This means, one program can start execution of another program by a RUN monitor call. The old program is lost. Arguments can be passed to the chained program through the communication-segment, which is "posted" intact to the new program. This facility allows a number of programs to work in sequence towards completion of a task.

3.2.4 COMMAND PROCESSOR AS A USER PROGRAM

The Command Processor is a special user program which runs when no other program is running.

The command processor obtains command lines from the user terminal. It parses these commands, and inserts default parameters before executing the command. The command processor may directly service the command using various system information monitor calls. Alternately it may chain to an appropriate program, passing arguments in the communication area. Typing directories and files, copying files, editing are all carried out by separate utility programs to which command processor chains.

Error messages are also displayed to the user by the command processor. When a user program is aborted, the system automatically chains to the command processor, passing it the appropriate error number. The command processor then displays the required error message.

3.3 OVERVIEW OF THE STRUCTURE AND THE WORKING OF THE OPERATING SYSTEM

Each user in the operating system is associated with a process. Besides these there are some system processes. Process is the basic functional unit in which operating system organises its activity. Operating system manages the processes such that they run independently of each other and interact only in a well defined manner. The single CPU is multiplexed between all the processes.

The processes are identified by three states, as shown in Figure 3.2.

(1) Run : Executing on the processor.

(2) Ready : Waiting for the processor.

(3) Blocked : Waiting for a resource.

All the "ready" processes are kept in a Ready-Queue (RDYQUE). One process is "running" at a time. This is called the Current Process (CP).

System has a hardware clock (ET) which interrupts the system at regular intervals. On each clock interrupt, the CPU is allocated to a new process. The state of the CP is saved and it is inserted at the end of the RDYQUE. The scheduler then selects a process for executing on CPU. The scheduling discipline used is First-In-First-Out. The selected process is made the current process, its state is restored (this involves loading various hardware registers

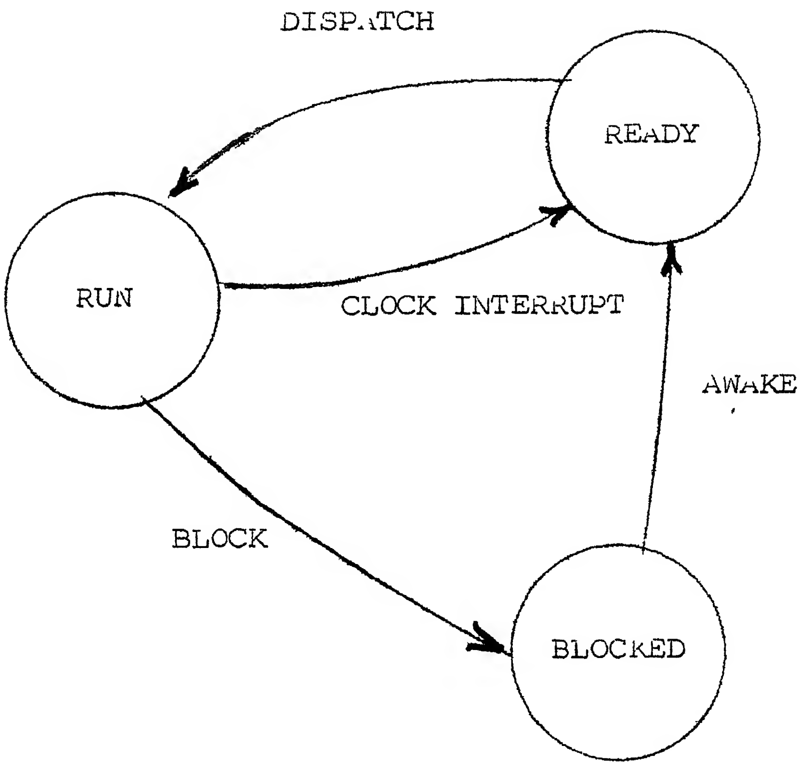


FIGURE 3.2 PROCESS STATE DIAGRAM

with appropriate values), and control is transferred to that program.

While running, a process makes monitor calls for obtaining system services. Some of these calls result in the current process getting blocked, as the service is not available. The state of the process is saved and its status is changed to "blocked". The scheduler is then called to select a new process for running.

When the service becomes available (generally indicated by an interrupt from the hardware device or by a signal operation by another process), the blocked process is awakened. The status of the process is changed to Ready and it is inserted at the end of the RDYQUE.

The process synchronization primitives provided are BINARY SEMAPHORES and EVENT BASED SIGNALS AND WAITS.

The operating system is designed as a hierarchy of three layers.

(1) The KERNEL is the innermost layer of the operating system. It provides the basic mechanisms for implementing processes, the processor allocation to these and synchronization between the processes.

It also provides the basic memory management functions for running each process in its own virtual space.

(2) The middle layer consists of Device-Subsystems and

Error-trap handlers.

A device subsystem consists of an interrupt routine to handle the device interrupts, and a set of service routines for obtaining logical services from the device.

The system processes are also implemented at this layer. System processes carry out those system functions which take a long time to complete. System processes are scheduled like user processes (e.g. Line Printer Spooler).

(3) The topmost layer contains the Monitor Call Handling routine.

This routine obtains the monitor call arguments provided by the user process and makes appropriate routine calls to the device subsystems for services. It also makes routine calls to the kernel for blocking the process and for other synchronisation functions.

All the above layers are executed in privileged mode using the Kernel Register Set. They run at highest priority and are uninterruptable. Thus they naturally constitute critical regions.

Besides these the operating system contains the file system routines, which are implemented as a supervisor. The supervisor has a shared data structure and consists of sharable

reentrant routines. These are used by various user processes concurrently. These procedures use the semaphore mechanism provided by the above three layers for implementing the critical regions and for synchronisation. These routines run in non-privileged mode using the user registers and get blocked like the user programs. The supervisor is implemented by simulating a supervisor mode.

The system also has a booting program which loads the operating system, initialises the system and brings in various utility programs into the main memory.

CHAPTER 4

STEPWISE DESIGN OF THE OPERATING SYSTEM

The PWS-316 operating system is designed as a set of modules. These modules form a hierarchy as shown in Figure 4.1. The relation defining the hierarchy is "routine call" [PARN 78].

The following sections describe the system modules, in a bottom-up fashion. Appendix A gives the program listings for each module.

4.1 PROCESS MANAGEMENT KERNEL

Process Management Kernel implements the processor allocation and process synchronization for a fixed number of processes, whose state information is always in main memory.

A process is implemented as a data structure called PROCESS-DESCRIPTOR. It contains the following information:

(1) Process Status (running, ready or blocked).

(2) Saved state of the process when not running.

(3) Memory allocation information and memory map for the currently executing program under this process.

(4) Blocking information: The cause of blocking and some associated parameters.

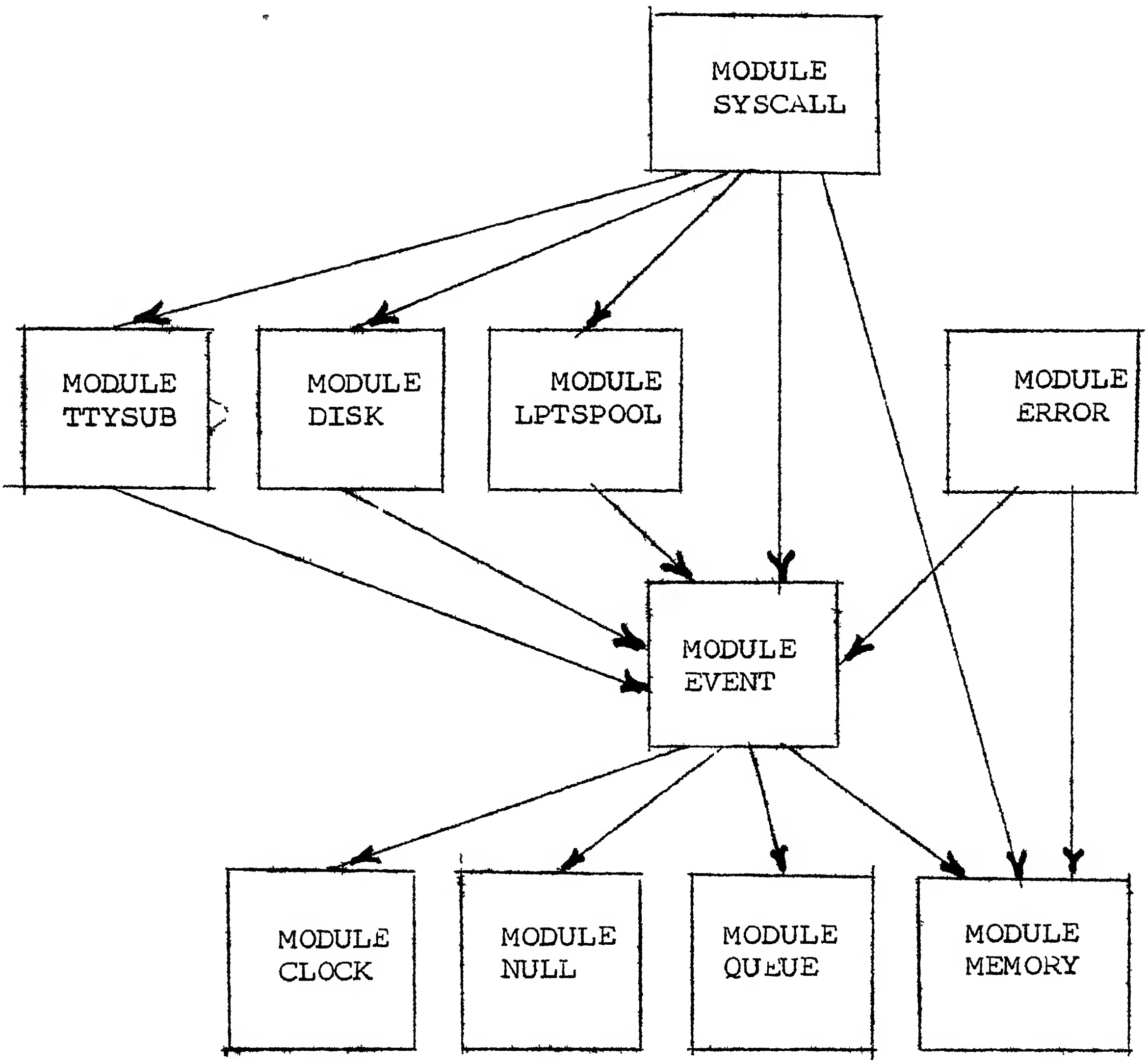


FIGURE 4.1 THE HIERARCHY OF MODULES

(5) System information:

(a) Flags indicating the resources assigned to the process.

(b) Current mode of the process: User or Supervisor

(c) Flag indicating whether D-space is enabled or not.

(6) The user information: name, user number, current directory pointer (CDP) etc.

(7) Links to insert the process in various process queues.

The module GLOBE defines the Process Descriptors.

The module QUEUE implements queue management for all process queues in the system. Routines INSERT and REMOVE are used to insert and remove processes from a queue, respectively. Routine ISEMPTY indicates whether a queue is empty or not.

RDYQUE is the queue of all "ready" processes.

Module CLOCK gives routines to manipulate the system clock. These allow initializing, starting, stopping and reading the clock.

Module EVENT is the heart of the process management kernel. This module itself is designed as a hierarchy of routines as shown in Figure 4.2 [WULF 75].

The innermost routines are SAVESTATE and LOADSTATE. Routine SAVESTATE saves the state of the Current Process (CP)

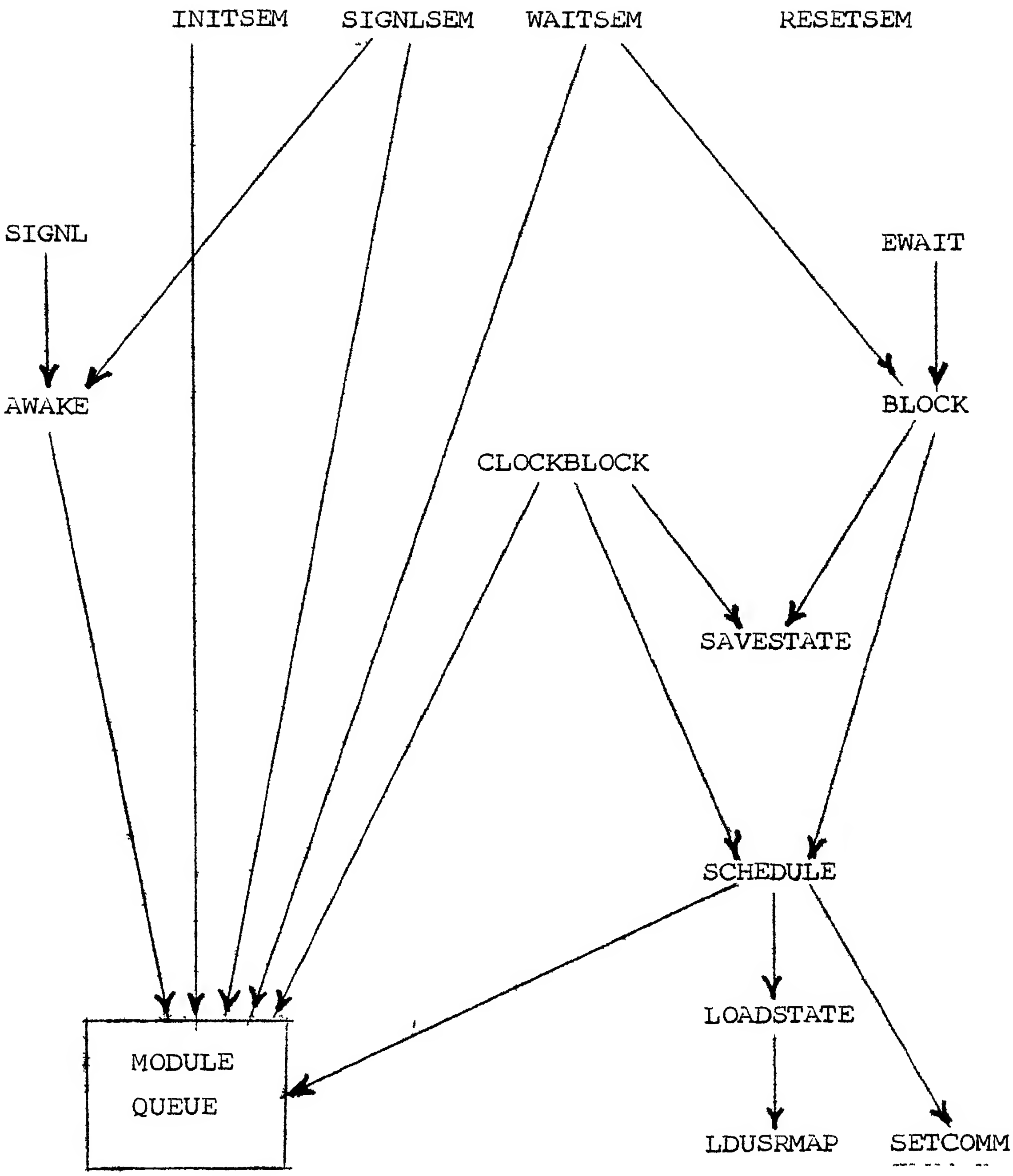


FIGURE 4.2 HIERARCHY OF ROUTINES IN PROCESS MANAGEMENT KERNEL

in its process descriptor. The routine LOADSTATE loads the state of the specified process onto the hardware. This includes:

(1) Loading GPR values.

(2) Loading User Map Registers with process memory map.

The next level consists of routines SCHEDULE and CLOCK-BLOCK.

The routine SCHEDULE selects a new process for running on CPU. Process at the head of RDYQUE is selected, giving First-In-First-Out scheduling policy. The state of this process is loaded and the process is "dispatched". The dispatch operation resets the system stack to empty and loads PC and PSW of the dispatched process on the hardware. It also enables the MAP unit appropriately.

The scheduler dispatches a null process, if RDYQUE is empty. Null process is always in "ready" state. Null process is implemented by the module NULL.

The routine CLOCK-BLOCK is an interrupt service routine, invoked at each Clock Interrupt. It saves the state of CP and inserts thus process in RDYQUE. Then it restarts the clock and calls SCHEDULE to dispatch a new process.

The routines above this level implement Process Synchronizaation primitives. The basic routines, BLOCK and AWAKE, have the following action.

Routine BLOCK saves the state of the specified process

and changes its status to "Blocked". The routine AWAKE inserts the specified process in RDYQUE and changes its status to "ready". Some operating system routines directly use primitives BLOCK and AWAKE for synchronization (e.g. disk manager).

The next level contains routines EWAIT and SIGNAL, which are EVENT based primitives for synchronoziation. Atmost one process can wait an event at a time. The routine EWAIT blocks the specified process, and marks it as "blocked" on the specified event. When the event occurs, a SIGNL operation is carried out. The routine SIGNL checks if the specified process is blocked on the specified event. If so, then the process is awakened, otherwise no action is taken.

The primitives EWAIT and SIGNL though not very general, are used quite frequently in this design as they have significantly lower overheads than semaphores.

The most general synchronization primitives implemented are BINARY SEMAPHORES. Each semaphore has a boolean flag and a queue of waiting processes associated with it.

Pseudo-code in Figure 4.3 shows the action of the semaphore primitives. These include INITSEM, RESETSEM, WAITSEM and SIGNLSEM.

A commonly used method for handling service requests from a user process is as follows:

(i) If service can be satisfied, it is completed. Otherwise, the requesting process is blocked using

```
routine INITSEM(SEMNO)=
   begin
    INITQUE(SEMQ[SEMNO]);FLG[SEMNO]:=TRUE
   end;

routine RESETSEM(SEMNO)=
   begin
    FLG[SEMNO]:=false
   end;

routine WAITSEM(SEMNO)=
   begin
    if FLG[SEMNO] then
       begin
             FLG[SEMNO]:=FALSE;
             RETURN
       end
    else
       begin
             INSERT(SEMQ[SEMNO],CP);
             BLOCK(CP)
       end
  end;

routine SIGNLSEM(SEMNO)=
    begin
     if ISEMPTY(SEMQ[SEMNO]) then
       FLG[SEMNO]:=true
     else
       begin
             AWAKE(FIRST(SEMQ[SEMNO]));
             RETURN
       end
    end;
```

FIGURE 4.3 BINARY SEMAPHORES

EWAIT primitive. The user's Program Counter is "instruction backed" by one instruction so that on awakening the process may retry for the service.

(ii) When service becomes available, a SIGNL operation is carried out. This awakens the process if blocked.

The awakened process then again requests for the service, which is now available.

4.2 MEMORY SYSTEM

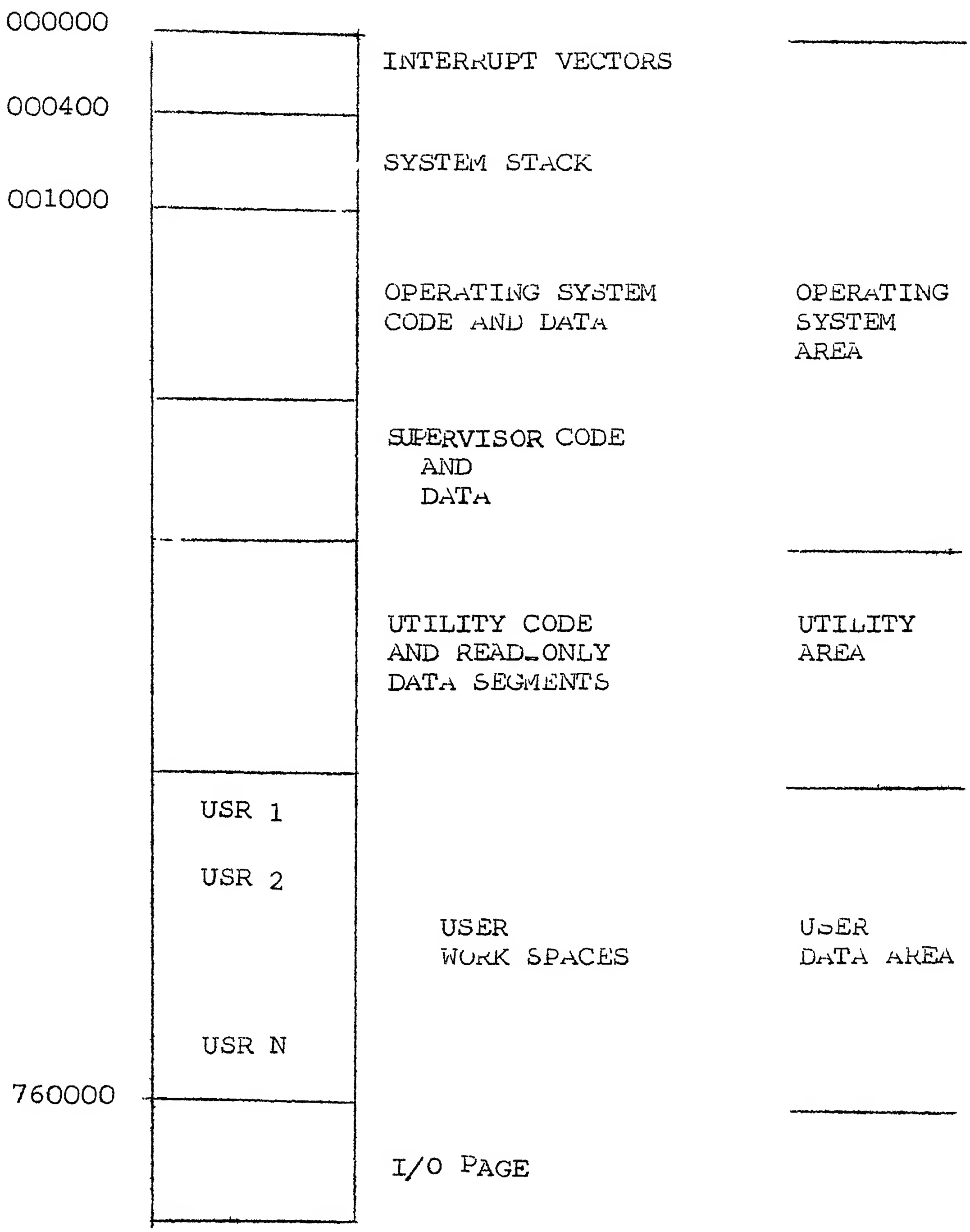
The Memory System allocates memory to the utility programs and the user data areas. It also maintains a table of available utilities and sets up the user's segment-maps when the user runs a new program.

The physical memory map of the system is shown in Figure 4.4. The TDC-316 hardware can support a maximum of 128K Word storage. Present configuration has 32K Words of storage.

The available memory is divided into 3 parts:

(1) The operating system area: This contains the operating system and supervisor. It also contains the system stack, the device interrupt vectors and the system processes.

(2) Utility area: Utilities in PWS-316 system are implemented as sharable, pure programs. All



PHYSICAL MEMORY MAP

FIGURE 4.4

utilities are resident in main memory and a single copy of their code is shared by all users. The utility area contains the code-segments and the read-only data segments of all programs.

(3) User data area: The remaining portion of main memory is called User Data Area. Presently the system has a fixed partition scheme for memory allocation in which User Data Area is equally divided between the user processes to form each user's Work Space. The data segments are set up from the user's work-space.

The system maintains a table of all utilities in the main memory called Active Program Table (ACTTAB). It contains the following information:

(1) Program name

(2) Virtual entrypoint of the program: PC is initialized to this value at the start of execution.

(3) Length of D-space required by the program: This excludes the stack segment and read-only data segments.

(4) Protection Code: This is used to restrict the availability of some programs only to some users.

(5) I-space-map for the program.

(6) Information for setting up D-space map: The length of each D-space segment and the address of read-only data segments.

The module MEMORY implements the memory system.

Routine LDUSRMAP loads the memory maps for the current user-program on the hardware. I-space map is taken from the ACTTAB while the D-space map is available in user's Process Descriptor.

Routine SETUSR is used to set up the user's process descriptor for execution of a new program. The D-space is set up in user's work-space by allocating memory to Data segments in sequence. Read-Only Data Segments are not set up, as a single copy of these is provided by the system, and is shared by all processes. A pointer to the appropriate I-map is also set up in the Process Descriptor.

The remaining space in user's work area is used to create the stack segment. The stack pointer is initialized to the top of this segment. The routine SETUSR also initializes the PC to the virtual entry point specified in the ACTTAB. The GPR's are initialized to 0. Further, the D-space enable bit in process Descriptor is turned on to signify that the process uses both I and D-spaces.

It is also possible to run a user's private programs by loading them in his work-space. The routine SETPRIV sets

up the I-space map for the loaded program, in the process-descriptor. The D-space enable bit is turned off to signify that the program uses only the I-space.

When program wishes to execute a new program, it makes a Run monitor-call, after placing the program name in Communication Area. The routine SCAN searches the ACTTAB table to find the appropriate entry. It also carries out the protection checks. A Linear Search is used as the table size is small.

Besides these, module MEMORY contains a few other routines . Routine SETCOMM is used by the schedules to make the Communication Segment of the Current Process available to the system as data-segment 6. The routine SETSUP is used to overlay the user virtual space by supervisor virtual space for running the filing system (See supervisor implementation).

The module SETACT implements loading of utility programs and setting up of ACTTAB table. It also allocates user-work space. These operations are done at the time of starting the system.

The utility programs are stored on the disk as files. These files are kept in a special directory owned by the privileged user. The first block of each file contains a Header Record. The Header Record contains all information required to set up an ACTTAB entry. It gives the name of

the utility, the length of D-space required by the utility, the lengths of all code segments and the lengths and nature (read-only or read-write) of each data segment. The rest of the file contains the utilities code segments and read-only data segments in load format.

The routine GETUTILITY opens the required utility file and sets up the ACTTAB entry for that utility using the Header Record. It also allocates memory space for the utility in utility area. It then loads the utility code-segments in contiguous memory blocks. Read-only data segments are also loaded alongwith.

At the time of starting the operating system, the booting program calls routine GETUTILITY with utility names provided by the operator.

After all the required utilities are loaded, the remaining area is equally divided among user processes. The process descriptor is loaded with appropriate values to reflect this assignment. The above function is carried out by the routine ALLOCUSR.

4.3 ERROR TRAPS

All the error traps result in the abortion of the current process. The user process returns to the command processor program.

The user PC at which abort occurs and the nature of

error are passed as parameters to the command processor. The command processor displays the appropriate error message.

The module ERROR implements these functions.

4.4 DEVICE SUBSYSTEMS

Device subsystems control the device hardware. They also supplement the functions provided by the device hardware to give user processes a logical view of the devices. For example, a device system may use buffering to allow overlapping of device operation and computation.

Each subsystem is implemented as a module having a private data structure (e.g. buffers, status words etc.). There is a set of conditions associated with the subsystem on which user processes can get blocked (e.g. input buffer empty). The sub-system recognises a set of predefined events which it signals. These signals awaken the blocked processes.

Each sub-system consists of three sets of routines:

(1) Device Driver: This routine initiates the action of the device.

(2) Device Interrupt Routine: This routine is executed when the device hardware indicates completion of some action by an interupt.

The routine manipulates the data structure (e.g. enter the received character in buffer or

change status) and may initiate the device action. It also recognises events and signals them using the SIGNL primitive provided by the kernel.

(3) Device Service Routines: The device service routines are used by the user processes to obtain the logical service from the sub-system (e.g. get a character from TTY input buffer).

These routines also manipulate the data structure. They may block the user process depending upon some conditions associated with the data structure.

Besides these there are error interrupt routines which handle errors from the device hardware. They indicate the error to the console and wait for corrective action to be taken. They may also abort processes wanting the service of the device.

Since both the device interrupt routine and the device service routines access the data structure concurrently, it is necessary that they run in "mutual exclusion". This is achieved by running both these routines at processor priority 7. Hence these routines are executed uninterrupted and act as critical sections.

4.4.1 TTY SUBSYSTEM

This subsystem handles the video terminals connected to the system through Asynchronous Line Controllers.

The TTY SUBSYSTEM maintains circular buffers for input and output on each line. Each line also has a status word. The subsystem consists of the device driver routine TRANSMIT, the device interrupt routine SCANNER and device service routines GETCH, PUTCH, ECHO, NOECHO, INEMPTY, OUTFUL, LINEIN, RESET, FLUSIN and FLUSOUT.

The subsystem recognises three conditions

(1) Input buffer empty

(2) Output buffer full

(3) No CRLF characters in input buffer.

These are used to block the user processes. The subsystem signals appropriate events when these conditions become false.

4.4.2 DISK SUBSYSTEM

The disk subsystem* accepts requests for reading and writing of disk sectors. The disk manager maintains a queue of all disk input-output requests and services them one after another.

The disk subsystem consists of the disk driver routine DRIVER, the disk interrupt routine DATATRANS and the service

---

* implemented by Aarti Kumar [KUMA 82].

routine ENTERREQUEST. Routine ENTERREQUEST is used by the user processes for requesting disk operations.

Each request results in the blocking of the requesting process using primitive BLOCK. On completion of the request, the interrupt routine DATATRANS awakens the blocked process using primitive AWAKE. The implementation details of this module can be found in [KUMA 82].

4.4.3 SYSTEM PROCESSES

The system services which get blocked in their execution and which take a long time to execute are implemented as system processes (e.g. swapper, loader, spoolers etc.). System processes are scheduled alongwith user processes.

System processes are allocated memory in the operating system area and they communicate with the operating system directly for transfer of data.

The current design contains a single system process, the Lineprinter Spooler*. Details of its implementation can be found in [KUMA 82].

4.5 SUPERVISOR MODE IMPLEMENTATION

The file system consists of a shared data structure and a set of shared reentrant routines. These routines are

---

* implemented by Aarti Kumar [KUMA 82].

used by the user processes to carry out various file system functions.

Critical regions are implemented in the filing system code, to maintain consistence of the shared data structure. Thus, user process may get blocked before entering a critical region. Also, when the file system requires some disk I/O, the user process gets blocked till the request is completed.

To implement blocking of processes executing the file system code, it would be preferable to run the file system code in user mode.

'It is common in some systems for supervisor code to be made part of each user process. User address space includes privileged code and data. One motivation for such design results from the fact that supervisor services take considerable amount of time. In this fashion, a user process in the midst of a supervisor call can be interrupted (or blocked) to allow other processing in a manner essentially the same as when user code is interrupted (or blocked). However, this leads to security flaws." [POPE 75]

One solution to this problem is to have a supervisor mode with its own virtual space and register set which is different from user mode and kernel mode. Then kernel can block processes in supervisor mode TOPS 10 . However TDC-316 hardware does not permit this.

In PWS-316 the supervisor mode is simulated by overlaying the supervisor virtual space over the user virtual space whenever the user makes a file system request. This can easily be done by loading the supervisor segment map onto the relocation hardware in place of the user map.

The overlaying of supervisor is done transparently to the user when he makes a request for a file system service using switch-to-supervisor monitor call. After loading the supervisor map, the monitor call handler passes control to the supervisor at a fixed entry point. The user and the supervisor modes share the communication segment and the stack segment in data-space. Arguments are passed through the stack while long file specifications are passed through the communication segment.

On completion of the service, supervisor makes a return-to-user call which restores the user's virtual space.

Module SUPERVISOR contains the code for the implementation of supervisor mode. The details of file system design can be found in [KUMA 82].

4.6 MONITOR CALL HANDLING

The user process makes monitor calls using the TRAP instruction. Arguments are passed through the user registers. Long arguments and results are passed through the communication area.

Module SYSCAL implements monitor call handling. The interrupt routine MONITOR is invoked whenever user makes a TRAP instruction. This routine makes routine calls to the various subsystems and uses kernel primitives for blocking and awakening the processes. The communication segment is available as segment 6 of the kernel D-space. (On each schedule this segment 6 is made to point to the seg 0 of the new CP, using routine SETCOMM).

4.7 BOOTING THE SYSTEM

The booting of PWS-316 operating system is carried out in two parts.

The first part consists of loading the operating system from a predefined area on the disk into the main memory. This is done by program LOAD. Program LOAD itself is read into the main memory from a paper tape. After loading, program LOAD transfers control to program BOOT in the operating system area.

Program BOOT carries out the following sequence of actions.

(1) All the modules of the operating system are initialized.

(2) Various utilities are brought into the main memory under operator's commands. The Active

Program Table (ACTTAB) is initialized and the User Work Areas are set up.

(3) The user process-descriptors are initialized to run the command processor program under the user number (0,0). This number signifies that the only operation allowed to the user is LOGIN.

(4) The processes are inserted into the RDYQUE and routine SCHEDULE is called to despatch a process.

CHAPTER 5

IMPLEMENTATION AND TESTINGS

The programming language BLISS-11 was selected for implementation of the PWS-316 project. BLISS-11 has many desirable features which make it suitable for an operating system implementation on TDC-316 [WULF 71].

(1) BLISS-11 allows access to all the hardware features, permitting the coding of entire operating system in BLISS-11.

(2) The object code generated by Bliss-11 does not require any run time support.

(3) It gives control over the representation of data structures and methods to access them.

(4) It has a flexible range of control structures (including recursion and coroutines). It encourages program structuring and readability.

(5) It allows modularization of the system into separately compilable submodules. Each module can have its private data and a set of routines. Some of these data and routines can be made global allowing access to them by other modules.

The modularization facility gives a powerful abstraction mechanisms and allows program structuring. It also facilitates debugging.

(6) The Bliss-11 compiler is very highly optimizing compiler.

(7) Cross software is available for using the code generated by Bliss-11 compiler on TDC-316 system.

The support facilities available for implementation of PWS-316 project consist of a BLISS-11 compiler, a cross-assembler[+1] giving TDC-316 object code and linkers LNKX-11 and NEWLNK. LNKX-11 accepts object code of separately completed modules and generates absolute object code in a linear virtual space. This linker was used to generate the operating system code. NEWLNK[+1] allows generation of sharable code by relocating the instruction and the data references into separate I and D-spaces. It also supports segmentation and allows grouping of logically related objects into sections. This software runs on the DEC-10 computer at our installation.

A facility to download the object programs from the DEC-10 system[+2] was also available. A machine level debugger call DEBUG was written as a part of this project. It gives

---

+1 Developed at NCSDCT, TIFR, Bombay.

+2 Developed by George Paul at Computer Centre, I.I.T. Kanpur.

facility to examine and load memory locations and to set breakpoints in the loaded programs.

The operating system was designed as a hierarchy of layers [DIJK 68, DIJK 72]. Each layer was implemented as a set of MODULES. The system was decomposed into modules following these criteria:

(1) The interaction between modules must be minimum. It must generally be in the form of routine calls.

(2) The routines working over common data are placed in the same module.

(3) Logically related functions are grouped into a module [BERG 81 , PARN 72].

First the specification of the system and its structure were decided. The system was then implemented bottom up. For each layer the modules were defined and these definitions were "filled up" with code.

The testing of the system was also done in bottom-up fashion, alongwith coding. Each module was tested by writing a test drives for the module. The debugger was used to trace the control flow and its effects. The bottom-up order of testing has the advantage that when a module is being tested, it may use the lower level modules which are already tested and available [ZELK 79].

The implementation of the PWS-316 operating system has not been fully completed. However the basic operating system mechanisms have been coded and tested. The process-management-kernel, the memory system, the device subsystems and basic monitor-call handler are available.

The unfinished portions of the operating system are outlined below:

(1) System management functions like Login, Creating new users, Accounting etc. have not been implemented.

(2) The booting program does not load the utilities from the disk. Instead they are downloaded from the DEC-10 system. This change affects the module BOOT and the module SETACT.

(3) Most of the utilities are not available. A primitive command processor and a set of demonstration programs have been written.

CHAPTER 6

CONCLUSION.

The implementation of PWS-316 system is not complete enough to allow any performance evaluation. However our experience with the design of the operating system has led us to the following conclusions.

(1) It is possible to develop operating systems in reasonable time periods, if proper methodology is followed in its design and implementation.

(2) A good implementation language and tools are indispensible in implementating such systems. The language BLISS-11 went a long way in reducing our programming burden, allowing us to concentrate on the system structure.

(3) The structure of the system strongly influences the ease of its implementation and testing. It also affects the system extensibility.

The present design of the PWS-316 operating system has few limitations. They are outlined below. The possible extensions to the system are also indicated.

(1) The system has a very primitive memory management

system. All the utilities are resident in main-memory. Also, the user work space is assigned by a fixed partition scheme at the time of starting the operating system. This scheme is very waste-ful from memory utilization point of view.

The memory management scheme for the system can be extended to include swapping of utility code. The allocation of data space for the user processes can also be done dynamically.

(2) The system has a single queue of ready processes. All the processes are kept at the same priority level.

With more system processes (for TDC-DEC communication, loading, swapping etc.), it may be desirable to have many queues according to the process priority. A different scheduling policy can also be implemented.

(3) The synchronization primitives implemented are quite restrictive. Some form of mailbox facility to pass messages alongwith the signals is desirable. Also it may be advantageous to use a uniform mechanism for all synchronization.

The get-resource and put-resource primitives suggested by Shaw can provide an adequate synchronization mechanism [SHAW 74, SHAW 75].

REFERENCES

1. [BENS 69] - Bensoussan, A. and C.T. Clingen, 'The Multics Virtual Memory', Second Symposium on Operating System Principles, (Oct 69), pp. 30-42.

2. [BERG 81] - Bergland, G.D., 'A Guided Tour of Program Design Methodologies', IEEE COMPUTER, 14, 10 (Oct 81), pp. 13-36.

3. [DIJK 68] - Dijkstra, E.W., 'The Structure of THE Multiprogramming System', CACM, 8, 9 (Sept 68), pp. 341-346.

4. [DIJK 72] - Dijkstra, E.W., 'The Humble Programmer', CACM, 15, 9 (Oct 72), pp. 859-866.

5. [DOLO 78] - Dolotta, T.A., R.C. Haight and J.R. Mashey, 'The Programmer's Work Bench', The Bell System Technical Journal, 157, 6 (July 78), pp. 2177-2200.

6. [KUMA 82] - Kumar, A., 'A Filing System and Spooler for the TDC-316 Time Sharing System', M.Tech. Thesis, Computer Science Department, I.I.T. Kanpur (1982).

7. [PARN 72] - Parnas, D.L., 'On the Criteria to be used in Decomposing Systems into Modules', CACM, 15, 12 (Dec 72), pp. 1053-1058.

8. [PARN 78] - Parnas, D.L., 'On a "Buzzword": Hierarchical Structure', in Gries, D.(ed.), Programming Methodology. Springer Verlag (1978), pp. 335-342.

9. [POPE 75] - Popek, G.J. and C.S. Kline, 'A Verifiable Protection System', Proceedings, International Conference on Reliable Software, (Apr 75), pp. 298-299.

10. [RITC 74] - Ritchie, D.M. and K. Thompson, 'The Unix Time Sharing System', CACM, 17, 7 (July 74), pp. 365-375.

11. [SHAW 74] - Shaw, A.C., The Logical Design of Operating Systems. Prentice Hall Inc. (1974), Chapter 7, pp. 166-202.

12. [SHAW 75] - Shaw, A.C., 'The Design of a Single Language Multiprogramming System', in Zelkowitz, Shaw and Gannon, <u>Principles of Software Engineering and Design</u>, Prentice Hall, Inc., (1975), pp. 179-225.

13. [TDC 1] - <u>System Manual TDC-316</u>, Vol. 1, E.C.I.L.

14. [TDC 2] - <u>System Manual TDC-316</u>, Vol. 2, E.C.I.L.

15. [TDC 3] - <u>System Manual TDC-316</u>, Vol. 3, E.C.I.L.

16. [WULF 71] - Wulf, W.A., D.B. Russel and A.N. Habermann, 'BLISS: A Language for Systems Programming', <u>CACM</u>, 14, 12 (Dec. 71), pp. 780-790.

17. [WULF 75] - Wulf, W.A., 'Structured Programming in the Basic Layers of an Operating System', in <u>Languages, Hierarchies and Interfaces</u>, Lecture Notes in Computer Science, 46 (1975), Springer Verlag, pp. 293-334.

18. [ZELK 79] - Zelkowitz, H.V., A.C. Shaw, and J.D. Gannon, <u>Principles of Software Engineering and Design</u>, Prentice Hall, Inc., (1979), Chapter 2, pp. 51-52.

```
     ***********************************************************
     *    MODULE   : PARAM.B11                                 *
     *    AUTHOUR  :  Paritosh K. Pandva.                      *
     *    DATE     :  24/7/82                                  *
     *    FUNCTION: This module defines all the system         *
     *               parameters.                               *
     ***********************************************************

:RO FILE CONTAINING ALL THE SYSTEM PARAMETERS.
,L SYSTEM PARAMETERS START WITH &#   %

)
;&DNUMBER=3$,
;&QNUM=3$,          !NO OF QUEUES IN SYSTEM=RDYQUE+SEMAPHORE QUEUES
;&RDYQUE=0$,        !NO OF READY QUEUE
;&NULLPROC=0$,      !NULL PROCESS NO.
;&STKLIM=#1000$,          !TOP LIMIT OF SUPERVISOR STACK
;&BLIM  =6$,        !NO OF CLASS-B EVENTS 0 TO $&BLIM-1
;&SEMNO =2$,        !NO. OF SEMAPHORES
;&SEMQ  =1$,        !START OF SEMPHORE QUEUES
;&SEMBLK=60$,       !EVENT NO. FOR SEMAPHORE BLOCK
;&NOLINES=8$,       !NO. OF TTYS CONNCTED 0 TO NOLINES-1
,&INBSZ=20$,        !SIZE OF TTY INPUT BUFFERS
,&OTBSZ=20$,        !SIZE OF TTY OUTPUT BUFFERS
,&TMPBSZ=20$,       !SIZE  OF TMP BUFFER IN TTY SUBSYSTEM
,&TTY(PROCNO)=PROCNO+2$,        !PROCESS 0 TO 4 ARE SYSTEM PROCESS
,&ITTY(LINE)=LINE-2$,    !PROCESS NO FOR GIVEN TTY LINE,

,&PRNO=4$,          !NO. OF PROGRAMS IN ACTIVE PROGRAM TABLE
,&COMM(ADDR)=(#140000+ADDR)$;    !ACCESS COMMUNICATION SEG.
```

```
*****************************************************************
*    MODULE    : DEFFIL                                         *
*    AUTHOUR   :  Paritosh K. Pandya.                           *
*    DATE      :  24/7/82                                       *
*    FUNCTION: This module defines the trap call and           *
*              event numbers.                                   *
*****************************************************************%

%DEFFILE ASSIGNING VALUES TO TRAPS AND EVENTS %
macro
    TRP0=0$,
    TRP1=1$,           !INEMPTY TTY
    TRP2=2$,           !GETCH TTY
    TRP3=3$,
    TRP4=4$,
    TRP5=5$,
    TRP6=6$,
    TRP50=50$,
    TRP51=51$,
    TRP55=55$,
    TRP56=56$,
    TRP57=57$,


    EVT0=3$,           !TTY INPUT BLOCK (GETCH)
    EVT1=4$,           !TTY OUTPUT BLOCK (PUTCH)
    EVT2=5$,           !WAIT FOR EOLN IN TTY INPUT BUFFER
    EVT60=60$;         !SEMAPHORE BLOCK
```

```
%        ***************************
         *    MODULE   :  GLOBE
         *    AUTHOUR  :  Paritosh K. P
         *    DATE     :  24/7/82
         *    FUNCTION:   This module d
         *                by other modu
         ***************************
module GLOBE=
    !GLOBAL DEFINITIONS AND INITIALIZ

    begin
        !Process descriptor definition
        !
        !MEMORY MANAGEMENT EXTENSION
        !UPDATE SEGMENTED MEMORY SYSTE

        %EXTERNAL MACRO
                        &&DNUMBER

        require PARAM.B11;

        macro
            !PROCESS DESCRIPTOR DETAIL
            DSIZE=80$,          !SIZE OF P

            !DEFINITION OF VARIOUS  FI
            JBSTS=0$,           !JOB STAT

            JBFLNK=2$,
            JBBLNK=4$,          !LINK FOR
            JBBLKID=6$,         !BLOCKING
            JBPC=8$,
            JBPS=10$,           !PROSSESO
            SUPPC=12$,          !PC AT US
            SUPPS=14$,
            JBTIME=16$,         !CPU TIME
            JBGPR=18$,                  !
            JBIPTR=32$,         !USER I S
            JBDSEG=34$,         !USER D S
            JBFLG=66$,          !FLAGS AS
            JBDORG=68$,         !ORIGIN O
            JBDLEN=70$,         !LENGTH O
            JBPPN=72$,          !USER P,P
            JBNAME=74$,         !USER NAM
            JBPRNAME=82$,       !NAME OF

            !FLAG FIELD DEFINITIONS
            SUPBIT=0,1$,        !USER/SUP
            DSPBIT=1,1$,        !D-SPACE

            !DEFINITION OF JOB STATUS
```

```
        RUNSTS=0$,
        RDYSTS=1$,
        BLKSTS=2$;


    structure DESC[JOB,FIELD]=(.DESC+.JOB*DSIZE+.FIELD);
    word global DESC PROCESS[DSIZE*#%DNUMBER/2];         !ARRAY OF
    !PROCESS DESCRIPTORS.

    global CP,OLEPC,OLEPS,JIFFY,MPCSR;
    global routine INIT08=
        begin
            incr I from 0 to (#%DNUMBER*DSIZE)-2 by 2 do
                (PROCESS+.I)=0
        end;
  end
eludom
```

```
%
        **************************************************************
        *    MODULE   : QUEUE                                          *
        *    AUTHOUR  :  Paritosh K. Pandya.                           *
        *    DATE     :  24/7/82                                       *
        *    FUNCTION: This module implements queue management         *
        *              functions for all process queues in the         *
        *              system.                                         *
        **************************************************************%
module QUEUE(NOLIST)=
   % Generalised queue handling procedures      2/5/82 P.K.PANDYA %
   begin
      require PROC02.B11;          !DEFINITION OF PROCESS DESCRIPTOR

      %EXTERNAL MACRO &&QNUM ; NO OF QUEUES  %
      require PARAM.B11;

      word own VECTOR QUEFST:QUELST:QUECOUNT[%&QNUM];
      global routine INITQUE(QUENO)=
         begin
            QUEFST[.QUENO]=QUELST[.QUENO]=-1;
            QUECOUNT[.QUENO]=0
         end;

      global routine INSERT(QUENO,PROCNO)=
         begin
            %INSERT GIVEN PROCESS AT THE END OF RDYQUEUE  %
            if .QUECOUNT[.QUENO] eql 0 then
               begin
                  QUEFST[.QUENO]=.PROCNO;
                  QUELST[.QUEVO]=.PROCNO;
                  QUECOUNT[.QUENO]=1;
                  PROCESS[.PROCNO,JBFLNK]=PROCESS[.PROCNO,JBBLNK]=-1
               end
            else
               begin
                  local TMP;
                  QUECOUNT[.QUENO]=.QUECOUNT[.QUENO]+1;
                  TMP=.OUFLST[.QUENO];
                  QUELST[.QUENO]=.PROCNO;
                  PROCESS[.PROCNO,JBFLNK]=-1;
                  PROCESS[.PROCNO,JBBLNK]=.TMP;
                  PROCESS[.TMP,JBFLNK]=.PROCNO
               end
         end;

      global routine FIRST(QUENO)=
         begin
            %OBTAIN THE FIRST ELEMENT OF RDYQUEUE AND REMOVE IT %
            if .QUECOUNT[.QUENO] eql 1 then
               begin
                  QUECOUNT[.QUENO]=0;
```

```
            return .QUEFST[.QU
        end
    else
        begin
            local TMP1,TMP2;
            QUECOUNT[.QUENO]=
            TMP1=.QUEFST[.QUE
            QUEFST[.QUENO]=.P
            TMP2=.QUEFST[.QUE
            PROCESS[.TMP2,JBB
            return .TMP1
        end
    end;

global routine ISEMPTY(QUENO)
    begin
        if .QUECOUNT[.QUENO] eq
            return -1
        else
            return 0
    end;


%GLOBAL ROUTINE SHOWQUE(QUENO
                        BEGIN
                            EXTERN
                            MACRO

                            PROMPT
                            IF .QU
                               MES
                            ELSE
                             BEGIN
                              LOCA
                              TMP=
                              INCR
                                 (D
                                  T
                             END
                        END;%

global routine INIT01=
    begin
        %INITIALZATION ROUTINE%
        incr I from 0 to %%QNUM
            INITQUE(.I)
    end;

  end
eludom
```

```
%       *************************************************************
        *    MODULE   : CLOCK                                       *
        *    AUTHUOR  :  Paritosh K. Pandya.                        *
        *    DATE     :  24/7/82                                    *
        *    FUNCTION: This module gives basic functions to         *
        *              handle the system clock.                     *
        *************************************************************%
module CLOCK(NOLIST)=
   % MODULE CLOCK          P.K.PANDYA      4/7/82%
   begin
      bind
          CKCSR=#777542,
          CKETL=#777540,
          CKETH=#777546,
          CKVEC=#134;

      macro
          IENB=6,1$,
          STET=8,1$,
          MODE=9,1$,
          REET=13,1$;

      external CLOCKBLOCK;          !FROM EVENT
      external JIFFY;

      global routine RESTCLK=
         begin
             CKCSR<STET>=1
         end;
      global routine STOPCLK=
         begin
             CKCSR<STET>=0
         end;
      global routine READCLK=
         begin
             CKCSR<REET>=1;
             return -(.CKETL)
         end;
      global routine STRTCLK=
         begin
             CKCSR=#1530
         end;
      global routine INIT04=
         begin
             CKVEC=CLOCKBLOCK;
             (CKVEC+2)=#240; !<SUP MODE ,REGO,PRDI 5>
             CKETH=#377;
             CKETL=-.JIFFY;
         end;

   end
```

eludom

```
%       *******************************************************
        *    MODULE   : NULL                                  *
        *    AUTHOUR  :  Paritosh K. Pandya.                  *
        *    DATE     :  24/7/82                              *
        *    FUNCTION: This module implements the null process.  *
        *******************************************************%
module NULL=
   begin
      bind VAL=#777776;   !REG 7 OF TDC

      %EXTERNAL MACRO
                  &&NULLPROC=0$; %
      require PARAM.B11;

      require PROC02.B11;

      global routine INIT03=
         begin
            switches UNSAFE;
            PROCESS[&&NULLPROC,JBPC]=NULL;
            PROCESS[&&NULLPROC,JBPS]=#1400;  !<USR MODE REG SET 1 PR
            %SET NULL PROC MAP%
            begin
               local TORG,TMP;
               TORG=0;
               TMP=PROCESS[&&NULLPROC,JBDSEG];
               incr I from 0 to 10 by 2 do
                  begin
                     (.TMP+.I)=#77407;      !READWRITE ACESS
                     (.TMP+16+.I)=.TORG;
                     TORG=.TORG+128
                  end;
               %SET I/O PAGE%
               (.TMP+14)=#77400;   !LF=127,ED=0,ACF=7
               (.TMP+16+14)=#7600
            end;
            %CLEAR REGISTER AND SET STACK%
            begin
               local TMP;
               TMP=PROCESS[&&NULLPROC,JBGPR];
               incr I from 0 to 12 by 2  do
                  (.TMP+.I)=0
            end;
            % OTHER INITIALIZATIONS <NAME,PPN ETC> %
         end;

      while 1 do
         VAL=.VAL+1
   end
eludom
```

```
%        ***********************************************************
         *    MODULE   : EVENT                                      *
         *    AUTHOUR  :  Paritosh K. Pandya.                       *
         *    DATE     :  24/7/82                                   *
         *    FUNCTION: This module implements tne process          *
         *              management kernel.It includes the           *
         *              scheduler,the clock interrupt routine       *
         *              and the process syncnronization primitives*
         ***********************************************************%
module EVENT(NOLIST)=
   % kernel process control functions    3/6/82 P.K.PANDYA%

   begin
      %EXTERNAL MACRO
                     &&RDYQUE          !READY QUEUE NO.
                     &&NULLPROC        !NULL PROCESS NO.
                     &&STKLIM          !SUPERVISOR STACK TOP LIMIT
                     &&BLIM            !CLASS B EVENTS &&ALIM TO &&BLIM
                     &&SEMNO           !NO OF SEMAPHORES 0 TO &&SEMNO-1
                     &&SEMBLK          !EVENT NO FOR SEMAPHORE BLOCK
                     &&SEMO            !START OF SEMAPHORE QUEUES

      require PARAM.B11;

      macro
          SEMONO(ARG)=ARG+&&SEMQS;
      bind
          MAPSR0=#774200; !MAP ENABLE REGISTER

      require PROC02.B11;
      external INITQUE,INSERT,FIRST,ISEMPTY; !FROM QUEUE
      external LDUSRMAP,SETCOMM,SETSUP;           !FROM MEMORY
      external READCLK,RESTCLK,STRTCLK,STOPCLK;
      external CP,OLEPC,OLEPS,JIFFY,MPCSR;        !FROM GLOBL.B11

      own FLG[&&SEMNO];   !SEMAPHORE FLAGS

      routine SAVESTATE=
         begin
            bind UREG1=#777762; !USER R1
            local TMP;

            TMP=PROCESS[.CP,JBGPR];
            incr I from 0 to 12 by 2 do
               (.TMP+.I)=.(UREG1+.I);        !SAVE USER REGISTERS R1
            PROCESS[.CP,JBPC]=.OLEPC;
            PROCESS[.CP,JBPS]=.OLEPS        !SAVE PC,PS
         end;
      routine LOADSTATE(PROCNO)=
         begin
            bind    UREG1=#777762;  !USER R1
```

```
        word local TMP1;

        TMP1=PROCESS[.PROCNO,JBGPR];
        incr I from 0 to 12 by 2 do
           (UREG1+.I)=.(.TMP1+.I);          !LOAD USER GPRS
        LDUSRMAP(.PROCNO);
        if .PROCESS[.PROCNO,JBFLG]<SUPBIT> then SETSUP(.PROCNO)
    end;


global routine SCHEDULE=
    begin

        if ISEMPTY(&&RDYQUE) then
           CP=&&NULLPROC
        else
           CP=FIRST(&&RDYQUE);      !SELECT A PROCESS FOR RUNNING
        LOADSTATE(.CP);            !CONVERT TO MACRO
        PROCESS[.CP,JBSTS]=RUNSTS;
        % <START CLOCK> %
        STRTCLK();
        %BORROW USER SEGMENT 0 FOR COMMUNICATION%
        SETCOMM(.CP);     !BORROW COMMUNICATION SEGMENT

        begin
           switches UNSAFE;
           MAPSR0=.MPCSR;   !ENABLE MAP AS REQUIRED
           &&STKLIM=.PROCESS[.CP,JBPS];
           (&&STKLIM-2)=.PROCESS[.CP,JBPC];
           SP=&&STKLIM-2;
           inline("RTI")     !DISPATCH
        end
    end;

global routine BLOCK(PROCNO)=
    begin
        SAVESTATE();      !CONVERT TO MACRO
        PROCESS[.PROCNO,JBSTS]=BLKSTS;
        % <ACCOUNTING> %
        PROCESS[.PROCNO,JBTIME]=.PROCESS[.PROCNO,JBTIME]+READCLK();STOPCLK(

        SCHEDULE()
    end;

global routine AWAKE(PROCNO)=
    begin
        !NAMED PROCESS IS PUT INTO &&RDYQUE
        PROCESS[.PROCNO,JBSTS]=RDYSTS;
        PROCESS[.PROCNO,JBBLKID]=-1;
        INSERT(&&RDYQUE,.PROCNO)
    end;
```

```
global routine EWAIT(PROCNO,EVENTNO)=
   begin

      if .EVENTNO lss &&BLIM then
         begin
            PROCESS[.PROCNO,JBBLKID]=.EVENTNO;
            BLOCK(.PROCNO)
         end
   end;

global routine SIGNL(PROCNO,EVENTNO)=
   begin

      if .EVENTNO lss &&BLIM then
         if .PROCESS[.PROCNO,JBSTS] eql BLKSTS then
            if .PROCESS[.PROCNO,JBBLKID] eql .EVENTNO then
               %THIS JOB IS BLOCKED ON THIS EVENT
               AWAKE(.PROCNO)
   end;

global routine INITSEM(NUMBER)=
   begin
      FLG[.NUMBER]=-1 ;INITQUE(SEMQNO(.NUMBER))
   end;

global routine RESETSEM(NUMBER)=
   FLG[.NUMBER]=0;

global routine WAITSEM(NUMBER)=
   begin
      if .FLG[NUMBER] then
         FLG[NUMBER]=0
      else
         begin
            INSERT(SEMQNO(NUMBER),.CP);
            PROCESS[.CP,JBBLKID]=&&SEMBLK;
            BLOCK(.CP)
         end
   end;
global routine SIGNLSEM(NUMBER)=
   begin
      if ISEMPTY(SEMQNO(.NUMBER)) then
         FLG[.NUMBER]=-1
      else
         begin
            byte local TMP;
            TMP=FIRST(SEMQNO(.NUMBER));
            AWAKE(.TMP)
         end
   end;
```

```
global routine INTERRUPT CLOCKBLOCK=
    begin
        OLEPC=.OLDPC;
        OLEPS=.OLDPS;
        PROCESS[.CP,JBSTS]=RDYSTS;
        PROCESS[.CP,JBTIME]=.PROCESS[.CP,JBTIME]+.JIFFY;
        if .CP neq $&NULLPROC then
            begin
                INSERT(#&RDYQUE,.CP);SAVESTATE()
            end;
        SCHEDULE()
    end;

global routine INIT02=
    begin
        %INITIALISE MODULE EVENT%
        incr I from 0 to $&SEMNO-1 do
            INITSEM(.I);
    end;

  end
eludom
```

```
byte global ARRAY INPBUF[&&NOLINES,&&INBSZ];
byte global ARRAY OUTBUF[&&NOLINES,&&OTBSZ];
byte global VECTOR
      INPFST:
      INPLST:
      OUTFST:
      OUTLST:
      INPCOUNT:
      OUTCOUNT[&&NOLINES];

external SIGNL;

% TTY SCANNER MAINTAINS A CIRCULAR QUEUE FOR INPUT AND A CIRCULAR
               QUEUE FOR OUTPUT FOR EACH LINE. FOR EACH QUEUE TWO PIONT
               FRONT AND BACK AND A COUNT OF NO. OF ELEMENTS IS MAINTAI
               EACH LINE HAS A STATUS WHICH SHOWS CONDITION OF LINE.
               i.e. ECHO,CTRL-C,FOLN ETC.        %

routine TRANSMIT=
  begin
    local TMP,TMIDX;
    incr JX from 0 to NOLN1 do
      if .OUTCOUNT[.JX]<0,8> neq 0 then
        if .STATUS[.JX]<FREEBIT> then
          begin
            TMIDX=.OUTFST[.JX]<0,8>;
            TMP  =.OUTBUF[.JX,.TMIDX]<0,8>;
            TMP<LINNO>=.JX<0,3>;
            TMRD=.TMP;
            STATUS[.JX]<FREEBIT>=0;
            OUTFST[.JX]<0,8>=.OUTFST[.JX]<0,8>+1;
            if .OUTFST[.JX]<0,8> eql &&OTBSZ then
                OUTFST[.JX]<0,8>=0;
            OUTCOUNT[.JX]<0,8>=.OUTCOUNT[.JX]<0,8>-1;
            if .OUTCOUNT[.JX]<0,8> eql (&&OTBSZ-1) then
                SIGNL(&&TTY(.JX),EVT1);       !OUT BUFFER NOT FULL
          end
  end;

routine INTERRUPT SCANNER=
  begin
    local IX,TMIDX;
    byte local TLIN,TCH;

    IX= -1;
    while .SYCS<BVAL> do

      begin
        IX=.IX+1;
        CHBUF[.IX]=.RDLS
```

```
        end;
     incr I from 0 to .IX do
        begin
           TLIN=.CHBUF[.I]<LINNO>;
           if .CHBUF[.I]<RSTS> then
              % A NEW CHARECTER FROM SOME TTY  %
              begin
                 TCH=.CHBUF[.I]<0,7>;
                 if .INPCOUNT[.TLIN]<0,8> eql %&INBSZ then
                    % <ERROR ACTION>    %
                 else
                    begin
                       TMIDX=.INPLST[.TLIN]<0,8>;
                       INPBUF[.TLIN,.TMIDX]<0,8>=.TCH;
                       INPLST[.TLIN]<0,8>=.INPLST[.TLIN]<0,8>+1;
                       if .INPLST[.TLIN]<0,8> eql &&INBSZ then
                          INPLST[.TLIN]<0,8>=0;
                       INPCOUNT[.TLIN]<0,8>=.INPCOUNT[.TLIN]<0,8>+1;
                       %.EVENT <INPBUF EMPTY>NONEMPTY>  %
                       if .INPCOUNT[.TLIN]<0,8> eql 1 then
                          SIGNL(%&ITTY(.TLIN),EVT0);   !INP BUUFER NO
                       % STATUS CHANGE   %
                       if .TCH eql CTRLC then
                          STATUS[.TLIN]<CTLCBIT>=1;
                       if .TCH eql EOLN then
                          (STATUS[.TLIN]<LINCNT>=.STATUS[.TLIN]<LINCN
                           if .STATUS[.TLIN]<LINCNT> eql 1 then
                              SIGNL(&&ITTY(.TLIN),EVT2) );    !EOLN

                       %ECHO CHARECTER    %
                       if .STATUS[.TLIN]<ECHOBIT> then
                          if .OUTCOUNT[.TLIN]<0,8> eql &&OTBSZ then
                             % <CANT ECHO THE CHARECTER >  %
                          else
                             begin
                                TMIDX=.OUTLST[.TLIN]<0,8>;
                                OUTBUF[.TLIN,.TMIDX]<0,8>=.TCH;
                                OUTLST[.TLIN]<0,8>=.OUTLST[.TLIN]<0,8
                                if .OUTLST[.TLIN]<0,8> eql &&OTBSZ the
                                   OUTLST[.TLIN]<0,8>=0;
                                OUTCOUNT[.TLIN]<0,8>=.OUTCOUNT[.TLIN]<
                             end
                    end
              end;
           if .CHBUF[.I]<TSTS> then
              STATUS[.TLIN]<FREEBIT>=1
        end;
     TRANSMIT()
   end;

global routine GETCH(LINE,VARADDR)=
```

```
    begin
       local TMIDX;
       TMIDX=.INPFST[.LINE]<0,8>;
       (.VARADDR)<0,8>=.INPBUF[.LINE,.TMIDX]<0,8>;
       if .INPBUF[.LINE,.TMIDX]<0,8> eql EOLN then           !REM <0,8>
          STATUS[.LINE]<LINCNT>=.STATUS[.LINE]<LINCNT>-1;
       INPFST[.LINE]<0,8>=.INPFST[.LINE]<0,8>+1;
       if .INPFST[.LINE]<0,8> eql &&INBSZ then
          INPFST[.LINE]<0,8>=0;
       INPCOUNT[.LINE]<0,8>=.INPCOUNT[.LINE]<0,8>-1;
       return
    end;

global routine PUTCH(LINE,CHAR)=
    begin
       local TMIDX;
       TMIDX=.OUTLST[.LINE]<0,8>;
       OUTBUF[.LINE,.TMIDX]<0,8>=.CHAR<0,8>;
       OUTLST[.LINE]<0,8>=.OUTLST[.LINE]<0,8>+1;
       if .OUTLST[.LINE]<0,8> eql &&OTBSZ then
          OUTLST[.LINE]<0,8>=0;
       OUTCOUNT[.LINE]<0,8>=.OUTCOUNT[.LINE]<0,8>+1;
       TRANSMIT();
       return          !RETURN WITH SUCCESS
    end;

global routine FLUSIN(LINE)=
    begin
       INPFST[.LINE]<0,8>=INPLST[.LINE]<0,8>=INPCOUNT[.LINE]<0,8>=0;
       STATUS[.LINE]<CTLCBIT>=0;
       STATUS[.LINE]<LINCNT>=0
    end;

global routine FLUSOUT(LINE)=
    begin
       %LOCAL TEM;
                                 IF .OUTCOUNT[.LINE]<0,8> EQL &&OTBSZ THEN TE
       OUTFST[.LINE]<0,8>=OUTLST[.LINE]<0,8>=OUTCOUNT[.LINE]<0,8>=0;
       STATUS[.LINE]<FREFBIT>=1;
       %IF .TEM NEQ 0 THEN
                                    SIGNL(&&ITTY(.LINE),EVT1)%
    end;

global routine ECHO(LINE)=
    STATUS[.LINE]<ECHOBIT>=1;

global routine NOECHO(LINE)=
    STATUS[.LINE]<ECHOBIT>=0;

global routine RESET(LINE)=
    begin
```

```
        %LOCAL TEM;
                                            IF .OUTCOUNT[.I
                                               TEM=-1;%
        INPFST[.LINE]<0,8>=INPLST[.LINE]<0,
        INPCOUNT[.LINE]<0,8>=OUTCOUNT[.LINE
        STATUS[.LINE]<FREEBIT>=1;
        STATUS[.LINE]<ECHOBIT>=1;
        %IF .TEM NEQ 0 THEN
                                               SIGNL(%&ITT)
    end;

  global routine INEMPTY(LINE)=
    .INPCOUNT[.LINE]<0,8> eql 0;

  global routine OUTFUL(LINE)=
    .OUTCOUNT[.LINE]<0,8> eql %OTBSZ;

  global routine LINEIN(LINE)=
    .STATUS[.LINE]<LINCNT> neq 0;

  global routine INIT05=
    begin
        CHVT=SCANNER;
        (CHVT+2)=#240; !PRIO=5,E=0,G=0,SUPE
        incr I from 0 to NOLN1 do
            RESET(.I);
        TMNC=#377;
        SYCS<BREN>=SYCS<TMEN>=SYCS<BFEN>=1
    end;
end
eludom
```

```
%       ****************************************************************
        *    MODULE   :  SYSCAL                                         *
        *    AUTHOUR  :  Paritosh K. Pandya.                            *
        *    DATE     :  24/7/82                                        *
        *    FUNCTION:   This module handles user's service             *
        *                requests.                                      *
        ****************************************************************%
module SYSCALL (NOLIST)=
    begin
        bind
            TRPVC=#10,
            ARG1=#777774,
            ARG2=#777772,
            ARG3=#777770,
            APG4=#777766;

        %EXTERNAL MACRO
                            &&COMM(ADDR)          !ACCESS COMMUNICATION
                            &&TTY   %GIVES TTYNO FOR GIVEN PROCESS%
        require PARAM.B11;
        require DEFFIL.B11;
        require PROC02.B11;
        external  CP,OLEPC,OLEPS;        !FROM GLOBFL.B11

        external BLOCK,AWAKE,FWAIT,SIGNL,WAITSEM,SIGNLSEM,SCHEDULE;
        %TTY SUBSYSTEM%
        external GETCH,PUTCH,FLUSIN,FLUSOUT,ECHO,NOECHO,RESET;
        external INEMPTY,OUTFUL,LINEIN,INIT05;

        %MEMORY SUBSYSTEM%
        external SETUSR,SETPRIV,SCAN,SETSUP,LDUSRMAP;
        external SUPSERVICE;

        routine INTERRUPT MONITR=
            begin
                switches UNSAFE;
                OLEPC=.OLDPC;OLEPS=.OLDPS;

                select .ARG1<0,8> of
                    nset
                        !Tty subsystem calls
                        TRP0:
                        !RESET LINE
                        begin
                            RESET(&&TTY(.CP));
                            ARG1=0;

                        end;
                        TRP1:
                        !INEMPTY
                        begin
```

```
        if INEMPTY(&&TTY(.CP)) then
            ARG1<0,8>=1
        else  ARG1<0,8>=0;
        ARG1<8,8>=0
    end;
    TRP2:
    !GETCH
    begin
        if INEMPTY(&&TTY(.CP)) then
            begin
                %BLOCK AFTER INSTRUCTION BACKING%
                OLEPC=.OLEPC-2;
                FWAIT(.CP,FVT0)
            end
        else
            begin
                local TMP;
                GETCH(&&TTY(.CP),TMP);
                TMP<8,8>=0;
                ARG1=.TMP;
            end
    end;
    TRP3:
    !SET ECHO
    begin
        ECHO(&&TTY(.CP));
        ARG1=0
    end;
    TRP4:
    !SET NOECHO

    begin
        NOECHO(&&TTY(.CP));
        ARG1=0
    end;
    TRP5:
    !PUTCH
    begin
        if OUTFUL(&&TTY(.CP)) then
            begin
                OLEPC=.OLEPC-2;
                FWAIT(.CP,EVT1)
            end
        else
            begin
                local TMP2;
                TMP2=.ARG1<8,8>;
                PUTCH(&&TTY(.CP),.TMP2);
                ARG1=0;
            end
    end;
```

```
TRP6:
!FLUSIN
begin
   FLUSIN(&&TTY(.CP));
   ARG1=0
end;
!Run class monitor calls
TRP50:
!SWITCHSUP
begin
   SETSUP(); %<LOAD SUPERVISOR MAP>%
   PROCESS[.CP,SUPPC]=.OLEPC;
   PROCESS[.CP,SUPPS]=.OLEPS;
   PROCESS[.CP,JBFLG]<SUPBIT>=1;
   %<LOAD SUPERVISOR ENTRY POINT>%
   OLDPC=SUPSERVICE;
   OLDPS=#1400;
end;
TRP51:
!RESTORE USR
begin
   LDUSRMAP(); %<RESTORE USR MAP>%
   OLDPC=.PROCESS[.CP,SUPPC];
   OLDPS=.PROCESS[.CP,SUPPS];
   PROCESS[.CP,JBFLG]<SUPBIT>=0;
end;
TRP55:
!Run(addr)
begin
   if .ARG2 lss #17770 then
      begin
         local TMP;
         TMP=SCAN(&&COMM(.ARG2));
         if .TMP geq 0 then
            begin
               !VALID CALL
               SETUSR(.CP,.TMP);
               AWAKE(.CP);
               SCHEDULE()
            end
      end;
   &&COMM(0)=1;          !ERROR IN CHAINING
   &&COMM(2)=1;
   SETUSR(.CP,0);        !GOTO COMMAND PROCESSOR
   AWAKE(.CP);
   SCHEDULE()
end;
TRP56:
!RUN A USERS LOADED PROGRAM
begin
   local TMP1;
```

```
                    TMP1=.PROCESS[.CP,JBDLEN];
                    SETPRIV(.CP,.ARG2,.TMP1,.ARG3);
                    &&COMM(0)=0;
                    AWAKE(.CP);
                    SCHEDULE()
                end;
                TRP57:
                !EXIT FROM A PROGRAM
                begin
                    &&COMM(0)=0;
                    SETUSR(.CP,0);           !COMMAND PROCESSOR
                    AWAKE(.CP);
                    SCHEDULE()
                end;
            tesn
        end;
    global routine INIT06=
        begin
            TRPVC=MONITR;
            (TRPVC+2)=#340;          !SUP MODE,GPR 0,PRIO 7
        end;
    end
eludom
```

```
%         ***************************************************************
          *    MODULE   :  BOOT                                         *
          *    AUTHOUR  :  Paritosh K. Pandya.                          *
          *    DATE     :  24/7/82                                      *
          *    FUNCTION:   This module boots the operating system       *
          *                after it is loaded.                          *
          ***************************************************************%

module BOOT(MAIN(150))=
    begin
        external INIT01,INIT02,INIT03,INIT04,INIT05;
        EXTRENAL INIT06,INIT07,INIT08;
        external INSERT,SCHEDULE,SETUSR;
        external LDKERMAP;

        require PARAM.B11;
        require PROC02.B11;

        INIT01();INIT02();INIT03();INIT04();INIT05();
        INIT06();INIT07();INIT08();       !Initalise all system module

        %Active program table must be set here. alsoUser work areas
                     are assigned here.%
        PROCESS[1,JBDORG]=#1200;
        PROCESS[1,JBDLEN]=#200;
        PROCESS[2,JBDORG]=#1400;
        PROCESS[2,JBDLEN]=#200;


        %Initialise the process descriptors to run the comd
                     processor. %
        SETUSR(1,0);SETUSR(2,0);

        %Insert the processes in ready queue. %
        INSERT(&&RDYQUE,1);INSERT(&&RDYQUE,2);

        LDKERMAP();
        %Dispatch the first job %
        SCHEDULE()

    end
eludom
```

```
%       ***************************************************************
        *    MODULE   :   SUPERVISOR                                  *
        *    AUTHOUR  :   Paritosh K. Pandya.                         *
        *    DATE     :   24/7/82                                     *
        *    FUNCTION:    This module implements the supervisor       *
        *                 mode.The fileing system runs in this        *
        *                 mode.                                       *
        ***************************************************************%
module SUPERVISOR=
   begin
      % FILE SYSTEM PRIVATE DATA DEFINITION%
      global routine SUPSERVICE(FN,ARG1,ARG2,ARG3,ARG4)=
         begin
            case .FN of
               set
                  %FILE SYSTEM ROUTINES COME HERE%
               tes
         end;
      global routine INIT10=
         begin
            %FILE SYSTEM INITIALISATION%
         end ;
   end
eludom
```

```
%       ***********************************************************
        *    MODULE   : USRMAC                                    *
        *    AUTHOUR  :  Paritosh K. Pandya.                      *
        *    DATE     :  24/7/82                                  *
        *    FUNCTION:   This module provides the macro           *
        *                definitions used by user programs to    *
        *                make  monitor calls.                     *
        ***********************************************************

%USR MACROS%
require DEFFIL.B11;
macro
    &TTRST=
    begin
        register VAL=1;
        VAL=0;  %RESET &TTYLINE
        TRAP(0)
    end$,
    &TTINEMT=
    begin
        register VAL=1;
        VAL=1;
        TRAP(0);
        if .VAL<0,8> then return -1
        else return 0
    end$,
    &TTGETCH(ADDR)=
    begin
        register VAL=1;
        VAL=2;
        TRAP(0);
        ADDR=.VAL<0,8>
    end$,
    &TTPUTCH(CHAR)=
    begin
        register VAL=1;
        VAL<8,8>=CHAR;
        VAL<0,8>=5;
        TRAP(0)
    end$,
    &TTECHO=
    begin
        register VAL=1;
        VAL=TRP3;
        TRAP(0)
    end$,
    &TTNOECHO=
    begin
        register VAL=1;
        VAL=TRP4;
        TRAP(0)
```

```
ends,
&TIFLUSIN=
begin
   register VAL=1;
   VAL=TRP6;
   TRAP(0)
ends,

!RUN CLASS MONTOR CALLS
&RUN(ADDR)=
begin
   register VAL1=1;
   register VAL2=2;
   VAL1=TRP55;
   VAL2=ADDR;
   TRAP(0)
ends,
&RUNPRIV(LEN,ENTRY)=
begin
   register VAL1=1;
   register VAL2=2;
   register VAL3=3;
   VAL1=TRP56;
   VAL2=LEN;
   VAL3=ENTRY;
   TRAP(0)
ends,
&EXIT=
begin
   register VAL=1;
   VAL=TRP57;
   TRAP(0)
ends;
```